Lab 03

Wall Following: Feedback PD Control Worksheet

Robot Name _____Walter_____

Team Member Name _____Yao Xiong_____

Team Member name _____Zhengyang Bi_____

Purpose

1. In your own words, state the purpose of Lab 03 in the following space. What behaviors are you implementing on the robot?

   Deploy a algorithm for following wall with PD feedback control on the mobile robotics.

Part 1 – Follow Wall (Layer 1)

Bang-Bang Control

2. Did you decide to drive your robot forward or backward, how did you decide? What were the pros and cons?

   We decided to drive our robot forward. It is easier for calculation of direction and distance for driving robots. Also, it is convenient for debugging. However, sometimes it cannot sensor for barrier exists in its invisible area.

3. How far and how long was the robot able to follow the wall between 4 and 6 inches without losing it?

   The robot could run 6 feet lost.

Proportional Control

4. What proportional gain did you use so that the robot followed the wall with regular oscillations?

   The kp is 3.

5. How far and how long was the robot able to follow the wall without losing it?

   The robot will not lose its path in 6 feet.

Proportional-Derivative Control

6. What derivative gain did you use so that the robot followed the wall with minimal oscillations and limited hitting?

The kd is 1/10.

7. How far and how long was the robot able to follow the wall without losing it?

   The robot follows the wall with 6 feet without losing it.

8. How did you modify the code so that the robot could detect outside corner or doorway?

   The robot arrives at the outside corner. The robot turns toward the corner. If there is a corner, the robot continues following the wall. If not, the robot loses the wall.

Part 2 – Avoid Obstacle (Layer 0)

9. How did you integrate avoid obstacle into the previous part?

   If sensors detect the obstacle (risk of collision), the state will be transferred to avoid obstacles. If the robot is between 0 and 2 inches from the wall, execute the runaway behavior. This state is the layer0. It is the most prior action.

10. How does your robot handle a stuck situation? Did the robot ever get stuck?

    The robot will turn to the direction where sensors detect no wall and obstacle. The robot will get stuck unless it is trapped in all directions. The robot will also get stuck when there are some obstacles in the blind area of this robot.

Part 3 – Random Wander (Layer 3)

11. Describe how the robot's random wander behavior worked and how you integrated it with wall following and avoid obstacles.

    The random wander state will be launched when there are no obstacles and walls sensors. If sensors detect the wall, state transferred into wall following. If sensors detect the obstacle, the state will be transferred to avoid obstacles.

Part 4 – Follow Center (Layer 2)

12. Describe how you add the follow center layer to the subsumption architecture that you're already built?

    If two sensors at left and right both detect the wall, the state will change to follow center. The program will ensure that the robot keeps the center of the walls.

Part 6 – Go To Goal

13. How did you keep track of the robot's progress around an obstacle and ensure that it was still making progress toward the goal from the current position?

    We begin by calculating the total distance the robot needs to travel. During normal movement, we use motor encoders to track the robot's progress along this path. When the robot encounters an obstacle and needs to avoid it, we temporarily pause the encoder tracking. Once the robot successfully navigates around the obstacle, we resume using the encoder to monitor its progress toward the goal.

Conclusions

14. How does what you implemented on the robot compare to what you planned to based upon your software design plan?

    The software design matches the Finite state machine.

15. When tuning the proportional controller and/or derivative controller, did the robot exhibit any oscillating, damping, overshoot or offset error? If so, how much?

    The robot appears some of the oscillations. However, the oscillations period for distance is 4 feet.

16. What were the results of the different P and D controller gains? How did you decide which one to use?

    Large kp (10) would make the oscillation have higher frequency and amplitude, which cause the robot to a zigzag route. When the kd is large, the system is more fragile to the outside disturbance. Finally, the kp is 3, the kd is 1/10.

17. How accurate was the robot at maintaining a distance between 4 and 6 inches?

    Although the robot will sometimes go out of the range from 4 to 6 inches, the accuracy of the robot is good.

18. Did the robot ever lose the wall?

    The robot never loses the wall.

19. Compare and contrast the performance of the *Wander* and *Avoid* behaviors compared to last week's lab.

    The wander and Avoid behaviors are the same as last week's lab. However, the wall will be recognized as an obstacle in last week's lab. We integrate these two behaviors as the sate into our program.

20. What was the general plan to implement the feedback control and subsumption architecture on the robot?

    The sensor will provide distance feedback. The ideal value of the distance minus the distance feedback is error. The PD controller will process the error, providing a corrected output for reducing error.

21. How could you improve the control architecture and/or wall following/follow center behaviors?

    The P controller will be toned to suitable value for short time response. The D will also be adjusted to proper value for lower peak and less oscillation.

22. How did you implement the finite state machine to integrate the various behaviors? Did you use any inhibition and suppression to create layers in this behavior?

    We use the state variable to keep track of different states. We use the sensor's data to change the states. For the layer0 of avoid obstacle, when this behavior is launched, the other behavior will be suppressed.

23. How did you keep track of the robot's state and as it switched between behaviors?

    We used the LED for tracking the change of the state. The state is represented by different combinations of yellow, red, and green. In the program, we use a state variable to track the state.

24. What did you learn? What did you observe? How could you improve your performance?

    We have learned how to use PD control for acquiring smooth paths in different assignments. We also learned how to write the code to follow the state diagram.

Appendix

Insert your properly commented and modular code in the appendix of the worksheet.

```
/*
  Walter-Lab03.ino
  Yao Xiong & Zhengyang Bi 2024/2/4

  This program implements various autonomous behaviors for a wheeled robot using stepper motors and distance sensors.
  The robot features reactive behaviors including obstacle avoidance, runaway response, and following behavior.
  It uses a dual-core architecture (M7 and M4) for parallel processing of sensor data and motor control.

  The primary functions created are:
  goToAngle - turns robot to specified absolute angle in degrees
  goToGoal - moves robot to specified x,y coordinate in inches
  forward, reverse - both wheels move with same velocity, same direction
  spin - both wheels move with same velocity opposite direction
  smartWanderBehavior - random movement with obstacle avoidance
  runawayBehavior - moves away from detected obstacles
  followBehavior - maintains constant distance from detected obstacle
  collideBehavior - emergency stop when obstacle detected
  followWallBehavior - Follows walls with consistent distance

  Sensor Integration:
  - 4 LIDAR sensors (front, back, left, right) for distance measurement
  - 2 wheel encoders for position tracking
  - PID control system for accurate movement

  Core Architecture:
  M4 Core - Handles sensor data collection and processing
  M7 Core - Manages movement control and behavioral algorithms
  Uses RPC (Remote Procedure Call) for inter-core communication

  Hardware Connections:
  Arduino pin mappings: https://docs.arduino.cc/tutorials/giga-r1-wifi/cheat-sheet#pins
  A4988 Stepper Motor Driver Pinout: https://www.pololu.com/product/1182

  Stepper Motor Control:
  digital pin 48 - enable PIN on A4988 Stepper Motor Driver StepSTICK
  digital pin 50 - right stepper motor step pin
  digital pin 51 - right stepper motor direction pin
  digital pin 52 - left stepper motor step pin
  digital pin 53 - left stepper motor direction pin

  Status LEDs:
  digital pin 13 - enable LED on microcontroller
  digital pin 5 - red LED in series with 220 ohm resistor
```

```
  digital pin 6 - green LED in series with 220 ohm resistor
  digital pin 7 - yellow LED in series with 220 ohm resistor

  Sensors:
  digital pin 18 - left encoder pin
  digital pin 19 - right encoder pin
  digital pin 8 - front LIDAR
  digital pin 9 - back LIDAR
  digital pin 10 - left LIDAR
  digital pin 11 - right LIDAR

  Constants:
  Robot Physical Parameters:
  - Wheel radius: 1.7 inches
  - Robot width: 9.0 inches
  - Steps per revolution: 800

  Movement Parameters:
  - Default step speed: 300 steps/sec
  - Maximum speed: 1500 steps/sec
  - Maximum acceleration: 10000 steps/sec^2

  Sensor Parameters:
  - Maximum LIDAR distance: 40 cm
  - Obstacle threshold: 10 cm

  Required Libraries:
  - AccelStepper: https://www.airspayce.com/mikem/arduino/AccelStepper/
  Install via:
  Sketch->Include Library->Manage Libraries...->AccelStepper->Include
  OR
  Sketch->Include Library->Add .ZIP Library...->AccelStepper-1.53.zip

  See PlatformIO documentation for proper way to install libraries in Visual Studio
*/

// includew all necessary libraries
#include <Arduino.h>      //include for PlatformIO Ide
#include <AccelStepper.h>  //include the stepper motor library
#include <MultiStepper.h>  //include multiple stepper motor library
#include <math.h>
#include "RPC.h"

// state LEDs connections
#define redLED 5          // red LED for displaying states
#define grnLED 6          // green LED for displaying states
#define ylwLED 7          // yellow LED for displaying states
```

```
#define enableLED 13       // stepper enabled LED
int leds[3] = { 5, 6, 7 };  // array of LED pin numbers

// define motor pin numbers
#define stepperEnable 48  // stepper enable pin on stepStick
#define rtStepPin 50      // right stepper motor step pin
#define rtDirPin 51       // right stepper motor direction pin
#define ltStepPin 52      // left stepper motor step pin
#define ltDirPin 53       // left stepper motor direction pin

AccelStepper stepperRight(AccelStepper::DRIVER, rtStepPin, rtDirPin);  // create instance of right
stepper motor object (2 driver pins, low to high transition step pin 52, direction input pin 53 (high means
forward)
AccelStepper stepperLeft(AccelStepper::DRIVER, ltStepPin, ltDirPin);   // create instance of left stepper
motor object (2 driver pins, step pin 50, direction input pin 51)
MultiStepper steppers;                                  // create instance to control multiple steppers at the
same time

#define stepperEnTrue false  // variable for enabling stepper motor
#define stepperEnFalse true  // variable for disabling stepper motor
#define max_speed 1500       // maximum stepper motor speed
#define max_accel 10000      // maximum motor acceleration

int pauseTime = 2500;  // time before robot moves
int stepTime = 500;    // delay time between high and low on step pin
int wait_time = 1000;  // delay for printing data

// define encoder pins
#define LEFT 0                  // left encoder
#define RIGHT 1                   // right encoder
const int ltEncoder = 18;           // left encoder pin (Mega Interrupt pins 2,3 18,19,20,21)
const int rtEncoder = 19;           // right encoder pin (Mega Interrupt pins 2,3 18,19,20,21)
volatile long encoder[2] = { 0, 0 };  // interrupt variable to hold number of encoder counts (left, right)
int lastSpeed[2] = { 0, 0 };         // variable to hold encoder speed (left, right)
int accumTicks[2] = { 0, 0 };        // variable to hold accumulated ticks since last reset

#define TO_LEFT -1  // direction variables to left
#define TO_RIGHT 1  // direction variables to right

#define CLOCKWISE 1         // direction variables for clockwise motion
#define COUNTERCLOCKWISE -1  // direction variables for counterclockwise motion

// Helper Functions
const float pi = 3.14159;         // pi
const float wheelRadius = 1.7;    // robot wheel radius in inches
const int stepsPerRevol = 800;    // robot wheel steps per revolution
const float robotWidth = 9.0;     // robot width in inches
```

```
const int defaultStepSpeed = 200;  // robot default speed in steps per second

const int PIDThreshold = 50;  // PID threshold
const int PIDkp = 1;          // PID proportional gain
const int encoderRatio = 20;  // ratio between the encoder ticks and steps

// State tracking
enum RobotState {
  NO_WALL,
  FOLLOWING_LEFT,
  FOLLOWING_RIGHT,
  FOLLOWING_CENTER,
  TOO_CLOSE,
  TOO_FAR,
  CLOSE_TO_RIGHT,
  CLOSE_TO_LEFT,
  TURNING_CORNER,
  RANDOM_WANDER,
  INSIDE_CORNER,
  OUTSIDE_CORNER,
  COLLIDE_BEHAVIOR,
  RUNAWAY_BEHAVIOR,
  FOLLOW_BEHAVIOR,
  INSIDE_DEADBAND,
};

RobotState currentState;

void updateLEDs() {
  digitalWrite(redLED, LOW);
  digitalWrite(ylwLED, LOW);
  digitalWrite(grnLED, LOW);

  if (currentState == FOLLOWING_RIGHT) {
    digitalWrite(redLED, HIGH);
    digitalWrite(ylwLED, HIGH);
  } else if (currentState == FOLLOWING_LEFT) {
    digitalWrite(grnLED, HIGH);
    digitalWrite(ylwLED, HIGH);
  } else if (currentState == FOLLOWING_CENTER) {
    digitalWrite(redLED, HIGH);
    digitalWrite(grnLED, HIGH);
    digitalWrite(ylwLED, HIGH);
  } else if (currentState == RANDOM_WANDER) {
    digitalWrite(grnLED, HIGH);
  } else if (currentState == TOO_CLOSE || currentState == CLOSE_TO_RIGHT) {
    digitalWrite(ylwLED, HIGH);
```

```
  } else if (currentState == TOO_FAR || currentState == CLOSE_TO_LEFT) {
    digitalWrite(redLED, HIGH);
  } else if (currentState == INSIDE_CORNER) {
    digitalWrite(redLED, HIGH);
    digitalWrite(grnLED, HIGH);
  } else if (currentState == COLLIDE_BEHAVIOR) {
    digitalWrite(redLED, HIGH);
  } else if (currentState == RUNAWAY_BEHAVIOR) {
    digitalWrite(ylwLED, HIGH);
  } else if (currentState == FOLLOW_BEHAVIOR) {
    digitalWrite(redLED, HIGH);
    digitalWrite(grnLED, HIGH);
  }
}

// random move maximum values
const int maxTurnAngle = 90;      // Maximum turn angle in degrees
const int maxDistanceMove = 12.0;  // Maximum move distance in inches

// lidar constant values
#define FRONT 0
#define BANK 1
#define LEFT 2
#define RIGHT 3
#define numOfSens 4

uint16_t wait = 100;
int16_t ft_lidar = 8;
int16_t bk_lidar = 9;
int16_t lt_lidar = 10;
int16_t rt_lidar = 11;

int16_t lidar_pins[4] = { 8, 9, 10, 11 };

int16_t leftSnr = 3;
int16_t rightSnr = 4;

const int MAX_LIDAR_DISTANCE = 40;

// Check for obstacles - adjust these thresholds based on your needs
const int OBSTACLE_THRESHOLD = 10;  // centimeters

// a struct to hold lidar data
struct lidar {
  // this can easily be extended to contain sonar data as well
  int front;
  int back;
```

```
  int left;
  int right;
  // this defines some helper functions that allow RPC to send our struct (I found this on a random
forum)
  MSGPACK_DEFINE_ARRAY(front, back, left, right);  //
https://stackoverflow.com/questions/37322145/msgpack-to-pack-structures
https://www.appsloveworld.com/cplus/100/391/msgpack-to-pack-structures
} lidarData;

struct lidar read_lidars() {
  return lidarData;
}

// a struct to hold sonar data
struct sonar {
  // this can easily be extended to contain sonar data as well
  int left;
  int right;
  // this defines some helper functions that allow RPC to send our struct (I found this on a random
forum)
  MSGPACK_DEFINE_ARRAY(left, right);  // https://stackoverflow.com/questions/37322145/msgpack-to-
pack-structures https://www.appsloveworld.com/cplus/100/391/msgpack-to-pack-structures
} sonarData;

struct sonar read_sonars() {
  return sonarData;
}

struct lidar lidar_data;
struct sonar sonar_data;

// interrupt function to count left encoder tickes
void LwheelSpeed() {
  encoder[LEFT]++;  // count the left wheel encoder interrupts
}

// interrupt function to count right encoder ticks
void RwheelSpeed() {
  encoder[RIGHT]++;  // count the right wheel encoder interrupts
}

void allOFF() {
  for (int i = 0; i < 3; i++) {
    digitalWrite(leds[i], LOW);
  }
}
```

```
// function to set all stepper motor variables, outputs and LEDs
void init_stepper() {
  pinMode(rtStepPin, OUTPUT);               // sets pin as output
  pinMode(rtDirPin, OUTPUT);                // sets pin as output
  pinMode(ltStepPin, OUTPUT);               // sets pin as output
  pinMode(ltDirPin, OUTPUT);                // sets pin as output
  pinMode(stepperEnable, OUTPUT);           // sets pin as output
  digitalWrite(stepperEnable, stepperEnFalse);  // turns off the stepper motor driver
  pinMode(enableLED, OUTPUT);               // set enable LED as output
  digitalWrite(enableLED, LOW);             // turn off enable LED
  pinMode(redLED, OUTPUT);                  // set red LED as output
  pinMode(grnLED, OUTPUT);                  // set green LED as output
  pinMode(ylwLED, OUTPUT);                  // set yellow LED as output
  digitalWrite(redLED, HIGH);               // turn on red LED
  digitalWrite(ylwLED, HIGH);               // turn on yellow LED
  digitalWrite(grnLED, HIGH);               // turn on green LED
  delay(pauseTime / 5);                     // wait 0.5 seconds
  digitalWrite(redLED, LOW);                // turn off red LED
  digitalWrite(ylwLED, LOW);                // turn off yellow LED
  digitalWrite(grnLED, LOW);                // turn off green LED

  stepperRight.setMaxSpeed(max_speed);      // set the maximum permitted speed limited by processor
and clock speed, no greater than 4000 steps/sec on Arduino
  stepperRight.setAcceleration(max_accel);  // set desired acceleration in steps/s^2
  stepperLeft.setMaxSpeed(max_speed);       // set the maximum permitted speed limited by processor
and clock speed, no greater than 4000 steps/sec on Arduino
  stepperLeft.setAcceleration(max_accel);   // set desired acceleration in steps/s^2
  steppers.addStepper(stepperRight);        // add right motor to MultiStepper
  steppers.addStepper(stepperLeft);         // add left motor to MultiStepper
  digitalWrite(stepperEnable, stepperEnTrue);  // turns on the stepper motor driver
  digitalWrite(enableLED, HIGH);            // turn on enable LED
}

// function prints encoder data to serial monitor
void print_encoder_data() {
  static unsigned long timer = 0;                   // print manager timer
  if (millis() - timer > 100) {                     // print encoder data every 100 ms or so
    lastSpeed[LEFT] = encoder[LEFT];                // record the latest left speed value
    lastSpeed[RIGHT] = encoder[RIGHT];              // record the latest right speed value
    accumTicks[LEFT] = accumTicks[LEFT] + encoder[LEFT];    // record accumulated left ticks
    accumTicks[RIGHT] = accumTicks[RIGHT] + encoder[RIGHT];  // record accumulated right ticks
    Serial.println("Encoder value:");
    Serial.print("\tLeft:\t");
    Serial.print(encoder[LEFT]);
    Serial.print("\tRight:\t");
    Serial.println(encoder[RIGHT]);
    Serial.println("Accumulated Ticks: ");
```

```
    Serial.print("\tLeft:\t");
    Serial.print(accumTicks[LEFT]);
    Serial.print("\tRight:\t");
    Serial.println(accumTicks[RIGHT]);
    encoder[LEFT] = 0;   // clear the left encoder data buffer
    encoder[RIGHT] = 0;  // clear the right encoder data buffer
    timer = millis();    // record current time since program started
  }
}

/*this function will reset the encoders*/
void resetEncoder() {
  encoder[LEFT] = 0;   // clear the left encoder data buffer
  encoder[RIGHT] = 0;  // clear the right encoder data buffer
}

/*this function will stop the steppers*/
void stopMove() {
  stepperRight.stop();  // Stop right motor
  stepperLeft.stop();   // Stop left motor
}

/*
  This function performs PID control to adjust the position of the robot's wheels.
  Inputs:
    - leftDistance: Target distance for the left wheel.
    - rightDistance: Target distance for the right wheel.
  The function calculates the error between the target and actual encoder values,
  applies a proportional control to determine the necessary movement adjustments,
  and sets the wheel speeds accordingly. The steppers are then run to the computed positions.
  If the error is within a specified threshold, the function stops the steppers.
*/
void PIDControl(int leftDistance, int rightDistance) {
  // Calculate the error between the target and actual encoder values
  long leftDistanceError = abs(leftDistance) - encoder[LEFT] * encoderRatio;
  long rightDistanceError = abs(rightDistance) - encoder[RIGHT] * encoderRatio;

  // Apply proportional control
  if (abs(leftDistanceError) > PIDThreshold || abs(rightDistanceError) > PIDThreshold) {
    long outputLeft = leftDistanceError * PIDkp;
    long outputRight = rightDistanceError * PIDkp;

    // Set the wheel speeds
    if (leftDistanceError < 0) {
      stepperLeft.setSpeed(-stepperLeft.speed());
    }
    if (rightDistanceError < 0) {
```

```
    stepperRight.setSpeed(-stepperRight.speed());
   }

   // Run the steppers
   stepperLeft.move(outputLeft);
   stepperRight.move(outputRight);
   steppers.runSpeedToPosition();
  }
  stopMove();
}

/**
 * Converts a length in inches to a number of steps for the robot.
 * @param length the length in inches to convert
 * @return the number of steps for the robot to move the given length
 */

int length2Steps(double length) {
  return round(stepsPerRevol * length / (2 * pi * wheelRadius));
}

/**
 * Spins the robot about its center by rotating both wheels in opposite directions.
 * @param direction the direction of the spin (TO_LEFT or TO_RIGHT)
 * @param angle the angle of the spin in degrees
 * @param speed the speed of the spin in steps per second
 */
void spin(int direction, double angle, int speed) {
  stepperLeft.setCurrentPosition(0);
  stepperRight.setCurrentPosition(0);
  if (direction != TO_LEFT && direction != TO_RIGHT && angle < 0)
    return;

  // calculate the turn length, using the circle formula
  double stepperMoveLength = pi * robotWidth * angle / 360;
  int stepperMovePos = length2Steps(stepperMoveLength);

  if (direction == TO_RIGHT) {
    stepperLeft.move(stepperMovePos);
    stepperLeft.setSpeed(speed);

    stepperRight.move(-stepperMovePos);
    stepperRight.setSpeed(-speed);
  } else if (direction == TO_LEFT) {
    stepperLeft.move(-stepperMovePos);
    stepperLeft.setSpeed(-speed);
```

```
   stepperRight.move(stepperMovePos);
   stepperRight.setSpeed(speed);
  }

  steppers.runSpeedToPosition();
}

/**
 * Moves the robot forward by a specified distance at a specified speed.
 * @param distance the distance in inches to move the robot
 * @param speed the speed in steps per second to move the robot
 */
void forward(double distance, int speed) {
  resetEncoder();
  int stepperMovePos = length2Steps(distance);
  stepperLeft.setCurrentPosition(0);
  stepperRight.setCurrentPosition(0);

  stepperLeft.move(stepperMovePos);   // move left wheel to relative position
  stepperRight.move(stepperMovePos);  // move right wheel to relative position

  stepperLeft.setSpeed(speed);   // set left motor speed
  stepperRight.setSpeed(speed);  // set right motor spee
  steppers.runSpeedToPosition();

  PIDControl(stepperMovePos, stepperMovePos);
}

/**
 * Moves the robot backward by a specified distance at a specified speed.
 * @param distance the distance in inches to move the robot backward
 * @param speed the speed in steps per second to move the robot backward
 * The function calls the forward() function with negative values of distance and speed.
 */

void reverse(double distance, int speed) {
  forward(-distance, -speed);
}

/**
 * Turns the robot to the absolute angle specified.
 * @param angle the absolute angle to turn to in degrees
 * @note Positive angles are to the left, negative angles are to the right
 */
void goToAngle(double angle) {
  // Serial.println("goToAngle function");
```

```
  // digitalWrite(grnLED, HIGH);  //turn on green LED
  // if angle larger than 0, turn left
  // if angle less than 0, turn right
  int direction = 0;
  if (angle > 0) {
    direction = TO_LEFT;
  } else if (angle < 0) {
    direction = TO_RIGHT;
  } else {
    return;
  }
  angle = abs(angle);
  spin(direction, angle, defaultStepSpeed);
}

/**
 * @brief Calculates the turn angle in degrees based on the given x and y coordinates.
 *
 * This function computes the angle in radians using the arctangent of y/x, then converts it to degrees.
 * It adjusts the angle based on the quadrant of the (x, y) point to ensure the correct angle is returned.
 *
 * @param x The x-coordinate.
 * @param y The y-coordinate.
 * @return The turn angle in degrees.
 */
double getTurnAngle(double x, double y) {
  double angleRadian = atan(y / x);

  double angleDegree = angleRadian * 180.0 / pi;

  if (x > 0 && y > 0) {
    angleDegree = angleDegree;
  } else if (x < 0 && y >= 0) {
    angleDegree = 180 + angleDegree;
  } else if (x < 0 && y < 0) {
    angleDegree = 180 + angleDegree;
  } else if (x > 0 && y < 0) {
    angleDegree = angleDegree;
  }

  return angleDegree;
}

/**
 * Moves the robot to the goal location (x, y) in inches.
 * @param x the x-coordinate of the goal in inches
 * @param y the y-coordinate of the goal in inches
```

```
 * @note The robot will first turn to the correct angle, then move forward to the goal
 */
void goToGoal(double x, double y) {
  // Serial.println("goToGoal function");
  digitalWrite(redLED, LOW);   // turn off red LED
  digitalWrite(grnLED, HIGH);  // turn on green LED
  digitalWrite(ylwLED, HIGH);  // turn on yellow LED

  double distance = sqrt(pow(x, 2) + pow(y, 2));
  double angleDegree = getTurnAngle(x, y);

  Serial.print("Angle: ");
  Serial.println(angleDegree);

  goToAngle(angleDegree);
  delay(1000);
  forward(distance, defaultStepSpeed);
}

/**
 * Turns the robot by moving each wheel a different distance at a different speed.
 * @param direction the direction of the turn (TO_LEFT or TO_RIGHT)
 * @param timeDelay the time in seconds to move each wheel
 * @param velocityDiff the difference in speed between the two wheels
 */
void turn(int direction, double timeDelay, int velocityDiff) {
  if (direction != TO_LEFT && direction != TO_RIGHT)
    return;
  int speedHigh = 500 + velocityDiff;
  int speedLow = 500;

  // calculate the distance for each wheel
  int distanceShort = round(speedLow * timeDelay);
  int distanceLong = round(speedHigh * timeDelay);

  if (direction == TO_RIGHT) {
    stepperLeft.move(distanceLong);
    stepperLeft.setSpeed(speedHigh);  // set left motor speed
    stepperRight.move(distanceShort);
    stepperRight.setSpeed(speedLow);  // set right motor speed
  } else if (direction == TO_LEFT) {
    stepperLeft.move(distanceShort);
    stepperLeft.setSpeed(speedLow);  // set left motor speed
    stepperRight.move(distanceLong);
    stepperRight.setSpeed(speedHigh);  // set right motor speed
  }
```

```
  steppers.runSpeedToPosition();
}

/**
 * Prints the random values generated for the angle and distance.
 * @param randAngle the random angle generated
 * @param randDistance the random distance generated
 */
void printRandomValues(int randAngle, int randDistance) {
  Serial.print("Random Values:\n\tAngle: ");
  Serial.print(randAngle);
  Serial.print("\tDistance: ");
  Serial.println(randDistance);
}

/**
 * Generates a random angle and distance for the robot to move.
 * The robot will turn to the random angle, then move forward the random distance.
 */
void randomWander() {
  currentState = RANDOM_WANDER;
  updateLEDs();

  int randAngle = random(-maxTurnAngle, maxTurnAngle);
  int randDistance = random(maxDistanceMove);
  randDistance = length2Steps(randDistance);

  goToAngle(randAngle);

  stepperLeft.move(randDistance);
  stepperRight.move(randDistance);

  stepperLeft.setSpeed(defaultStepSpeed);
  stepperRight.setSpeed(defaultStepSpeed);
}

// reads a lidar given a pin
int read_lidar(int pin) {
  // Wait for pulse
  int16_t t = pulseIn(pin, HIGH);

  int d = MAX_LIDAR_DISTANCE;  // Default to "no object"
  if (t != 0 && t <= 1850) {
    d = (t - 1000) * 3 / 40;
    if (d < 0)
      d = 0;
  }
```

```
  delay(10);  // More stable than delayMicroseconds
  return d;
}

// reads a sonar given a pin
int read_sonar(int pin) {
  float velocity((331.5 + 0.6 * (float)(20)) * 100 / 1000000.0);
  uint16_t distance, pulseWidthUs;

  pinMode(pin, OUTPUT);
  digitalWrite(pin, LOW);
  digitalWrite(pin, HIGH);          // Set the trig pin High
  delayMicroseconds(10);            // Delay of 10 microseconds
  digitalWrite(pin, LOW);           // Set the trig pin Low
  pinMode(pin, INPUT);              // Set the pin to input mode
  pulseWidthUs = pulseIn(pin, HIGH);  // Detect the high level time on the echo pin, the output high level
time represents the ultrasonic flight time (unit: us)
  distance = pulseWidthUs * velocity / 2.0;
  if (distance < 0 || distance > 50) {
    distance = 0;
  }
  return distance;
}

// prints the sensor data
void print_sensor_data(struct lidar data, struct sonar data2) {
  Serial.print("lidar: ");
  Serial.print(data.front);
  Serial.print(", ");
  Serial.print(data.back);
  Serial.print(", ");
  Serial.print(data.left);
  Serial.print(", ");
  Serial.print(data.right);
  Serial.println();
  // Serial.print("sonar: ");
  // Serial.print(data2.left);
  // Serial.print(", ");
  // Serial.print(data2.right);
  // Serial.println();
}

// converts cm to inches
double cm2inch(int cm) {
  return 0.393701 * cm;
}
```

```
// collide behavior
void collideBehavior() {
  currentState = COLLIDE_BEHAVIOR;
  updateLEDs();
  stopMove();
  delay(50);  // Small delay to prevent CPU hogging
}

// check if there is an obstacle around the robot
bool isCloseObstacle() {
  // Read sensor data
  lidar_data = RPC.call("read_lidars").as<struct lidar>();
  sonar_data = RPC.call("read_sonars").as<struct sonar>();

  // Return true if any sensor detects a close obstacle
  return frontHasObstacle() || backHasObstacle() || leftHasObstacle() || rightHasObstacle();
  // return (sonar_data.left < OBSTACLE_THRESHOLD && sonar_data.left != 0) ||
  //      (sonar_data.right < OBSTACLE_THRESHOLD && sonar_data.right != 0);
}

// check if there is an obstacle in front of the robot
bool checkFrontObstacle() {
  lidar_data = RPC.call("read_lidars").as<struct lidar>();
  return frontHasObstacle();
}

// check if there is an obstacle in the front of the robot
bool frontHasObstacle() {
  return lidar_data.front < OBSTACLE_THRESHOLD;
}

// check if there is an obstacle in the back of the robot
bool backHasObstacle() {
  return lidar_data.back < OBSTACLE_THRESHOLD;
}

// check if there is an obstacle on the left of the robot
bool leftHasObstacle() {
  return lidar_data.left < OBSTACLE_THRESHOLD;
}

// check if there is an obstacle on the right of the robot
bool rightHasObstacle() {
  return lidar_data.right < OBSTACLE_THRESHOLD;
}
```

```
// Runaway behavior
const double runawayPropotion = 0.5;
const int forwardDistance = 10;

/**
 * @brief Executes the runaway behavior for the robot.
 *
 * This function controls the robot's movement based on sensor data to avoid obstacles.
 * It reads lidar data, calculates the necessary turn angle, and moves the robot accordingly.
 *
 * The function performs the following steps:
 * 1. Turns on the yellow LED and turns off the red and green LEDs.
 * 2. Reads lidar data using an RPC call.
 * 3. Calculates the x and y distances in inches based on the lidar data.
 * 4. Determines the turn angle based on the presence of obstacles detected by the sensors.
 * 5. Moves the robot to the calculated angle and moves forward if the turn angle is valid.
 * 6. Stops the robot if the turn angle is invalid.
 *
 * The function uses the following helper functions:
 * - frontHasObstacle(): Checks if there is an obstacle in front of the robot.
 * - backHasObstacle(): Checks if there is an obstacle behind the robot.
 * - leftHasObstacle(): Checks if there is an obstacle to the left of the robot.
 * - rightHasObstacle(): Checks if there is an obstacle to the right of the robot.
 * - getTurnAngle(double x_inch, double y_inch): Calculates the turn angle based on x and y distances.
 * - goToAngle(double angle): Turns the robot to the specified angle.
 * - forward(double distance, int speed): Moves the robot forward by the specified distance at the given
speed.
 * - stopMove(): Stops the robot's movement.
 */
void runawayBehavior() {
  currentState = RUNAWAY_BEHAVIOR;
  updateLEDs();

  lidar_data = RPC.call("read_lidars").as<struct lidar>();

  int x = lidar_data.front - lidar_data.back;
  int y = lidar_data.left - lidar_data.right;

  double x_inch = runawayPropotion * cm2inch(x);
  double y_inch = runawayPropotion * cm2inch(y);

  // print_sensor_data(lidar_data, sonar_data);

  double turnAngle = 0;

  if (!frontHasObstacle() && !backHasObstacle() && leftHasObstacle() && rightHasObstacle()) {
    turnAngle = 0;
```

```
  } else if (!leftHasObstacle() && !rightHasObstacle() && frontHasObstacle() && backHasObstacle()) {
    turnAngle = 90.0;
  } else if (frontHasObstacle() && backHasObstacle() && leftHasObstacle() && rightHasObstacle()) {
    turnAngle = 3600;
  } else if (!frontHasObstacle() && !backHasObstacle() && !leftHasObstacle() && !rightHasObstacle()) {
    return;
  } else {
    turnAngle = getTurnAngle(x_inch, y_inch);
  }

  if (turnAngle <= 360) {
    goToAngle(turnAngle);
    forward(forwardDistance, defaultStepSpeed);
  } else {
    stopMove();
  }
}

const double followPropotion = 0.25;
const double OBSTACLE_MARGIN = 2.0;
const double FOLLOW_SPEED = 200;

// Check if the robot is within the obstacle margin
bool isWithinObstacleMargin(double frontDistance) {
  return frontDistance < OBSTACLE_THRESHOLD + OBSTACLE_MARGIN && frontDistance >
OBSTACLE_THRESHOLD - OBSTACLE_MARGIN;
}

/**
 * @brief Executes the follow behavior for a robot using LIDAR data.
 *
 * This function controls the robot to follow an object by maintaining a certain distance
 * from it. It uses LIDAR data to determine the distance to the object and adjusts the
 * robot's movement accordingly. The function continuously reads LIDAR data and moves
 * the robot forward or backward to maintain the desired distance.
 *
 * The function performs the following steps:
 * 1. Turns on the red and green LEDs and turns off the yellow LED.
 * 2. Reads the initial LIDAR data and calculates the front error in centimeters and inches.
 * 3. Enters an infinite loop where it:
 *    - Checks if the object is within the obstacle margin.
 *    - If the object is not within the obstacle margin and is within the maximum LIDAR distance:
 *      - Calculates the movement distance based on the proportional control.
 *      - Moves the robot forward if the front error is positive, or backward if the front error is negative.
 *    - Reads the updated LIDAR data and recalculates the front error.
 *    - Delays for 20 milliseconds before the next iteration.
 *
```

```
 * The function exits the loop and stops the behavior if the object is beyond the maximum LIDAR
distance.
 */
void followBehavior() {
  currentState = FOLLOW_BEHAVIOR;
  updateLEDs();

  lidar_data = RPC.call("read_lidars").as<struct lidar>();
  int front_error_cm = lidar_data.front - OBSTACLE_THRESHOLD;
  double front_error_inch = cm2inch(front_error_cm);

  while (true) {
    if (!isWithinObstacleMargin(lidar_data.front)) {
      if (lidar_data.front >= MAX_LIDAR_DISTANCE)
        break;

      double moveDistance = abs(followPropotion * front_error_inch);

      if (front_error_cm > 0) {
        forward(moveDistance, FOLLOW_SPEED);
      } else if (front_error_cm < 0) {
        reverse(moveDistance, FOLLOW_SPEED);
      }
    }

    lidar_data = RPC.call("read_lidars").as<struct lidar>();
    front_error_cm = lidar_data.front - OBSTACLE_THRESHOLD;
    front_error_inch = cm2inch(front_error_cm);
    delay(20);
  }
}

/**
 * @brief Executes the smart wander behavior for a robot.
 *
 * This function makes the robot move in a random direction and distance,
 * while continuously checking for obstacles. If an obstacle is detected,
 * the robot will stop, perform a collision avoidance behavior, and then
 * resume wandering.
 *
 * The function performs the following steps:
 * 1. Generates a random angle and distance for the robot to move.
 * 2. Converts the distance to steps for the stepper motors.
 * 3. Rotates the robot to the random angle.
 * 4. Moves the robot forward by the random distance.
 * 5. Sets the speed of the stepper motors to the default speed.
 * 6. Continuously runs the stepper motors and checks for obstacles.
```

```
 * 7. If an obstacle is detected, stops the motors, performs collision
 *    avoidance, and resumes wandering.
 * 8. Turns on the green LED while moving, and turns off other LEDs.
 * 9. Adds a small delay to allow sensor readings to be processed.
 */
void smartWanderBehavior() {

  randomWander();

  while (steppers.run()) {
    digitalWrite(grnLED, HIGH);  // turn on green LED
    digitalWrite(ylwLED, LOW);
    digitalWrite(redLED, LOW);

    while (isCloseObstacle()) {
      // Obstacle detected - stop motors and turn on LED
      collideBehavior();
      delay(100);
      runawayBehavior();
    }
  }

  // Small delay to allow sensor readings to be processed
  delay(1);
}

/**
 * @brief Executes the smart follow behavior for a robot.
 *
 * This function makes the robot move in a random direction and distance,
 * while continuously checking for obstacles. If an obstacle is detected,
 * the robot will stop and execute the collide behavior, followed by the
 * follow behavior.
 *
 * The function performs the following steps:
 * 1. Generates a random angle and distance for the robot to move.
 * 2. Converts the distance to steps for the stepper motors.
 * 3. Rotates the robot to the generated angle.
 * 4. Moves the robot forward by the generated distance.
 * 5. Sets the speed of the stepper motors to the default speed.
 * 6. Continuously runs the stepper motors while checking for obstacles.
 * 7. If an obstacle is detected, stops the motors and executes the collide behavior.
 * 8. Turns on the green LED to indicate movement.
 * 9. Adds a small delay to allow sensor readings to be processed.
 */
void smartFollowBehavior() {
  randomWander();
```

```
  while (steppers.run()) {
    digitalWrite(grnLED, HIGH);  // turn on green LED
    digitalWrite(ylwLED, LOW);
    digitalWrite(redLED, LOW);

    while (checkFrontObstacle()) {
      // Obstacle detected - stop motors and turn on LED
      collideBehavior();
      delay(100);
      followBehavior();
    }
  }

  // Small delay to allow sensor readings to be processed
  delay(1);
}

/**
 * @brief Controls the movement and obstacle-following behavior of the robot.
 *
 * This function initializes the LEDs and stepper motor speeds, then enters an infinite loop
 * where it continuously checks for close obstacles. If an obstacle is detected, it calls
 * the collideBehavior() and followBehavior() functions to handle the obstacle. Once the
 * obstacle is removed, it resets the LEDs and stepper motor speeds and continues moving
 * the steppers one step at a time with a small delay to allow sensor readings to be processed.
 */
void moveAndFollowBehavior() {
  // Set initial movement speeds
  stepperLeft.setSpeed(defaultStepSpeed);
  stepperRight.setSpeed(defaultStepSpeed);

  while (true) {
    while (isCloseObstacle()) {
      collideBehavior();
      followBehavior();
    }
    // Obstacle removed - reset movement
    stepperLeft.setSpeed(defaultStepSpeed);
    stepperRight.setSpeed(defaultStepSpeed);

    // Move the steppers one step at a time
    stepperLeft.runSpeed();
    stepperRight.runSpeed();

    // Small delay to allow sensor readings to be processed
    delay(1);
```

```
  }
}

// Constants for wall following behavior
const double WallFollowKp = 3.0;    // Proportional gain - adjust this between 1-10
const double WallFollowKd = 0.1;    // Derivative gain
unsigned long lastMeasureTime = 0;  // For derivative calculation
double lastError = 0;               // For derivative calculation

const double TARGET_DISTANCE = 5.0;      // Target distance from wall (inches)
const double DEADBAND_INNER = 4.0;       // Inner deadband boundary (inches)
const double DEADBAND_OUTER = 6.0;       // Outer deadband boundary (inches)
const int FOLLOW_WALL_BASE_SPEED = 300;  // Base motor speed

// Convert inches to cm for lidar readings
const double DEADBAND_INNER_CM = DEADBAND_INNER * 2.54;
const double DEADBAND_OUTER_CM = DEADBAND_OUTER * 2.54;
const double TARGET_DISTANCE_CM = TARGET_DISTANCE * 2.54;

const int WALL_THRESHOLD = 35;
const bool bangBangControllorOn = true;

bool leftHasWall() {
  return lidar_data.left < WALL_THRESHOLD;
}

bool rightHasWall() {
  return lidar_data.right < WALL_THRESHOLD;
}

bool frontHasWall() {
  return lidar_data.front < WALL_THRESHOLD;
}

bool isLeftCorner() {
  return frontHasWall() && leftHasWall() && !rightHasWall();
}

bool isRightCorner() {
  return frontHasWall() && rightHasWall() && !leftHasWall();
}

void avoidObstacle() {
  collideBehavior();
  runawayBehavior();
}
```

```
void followWallBehavior() {
  randomWander();
  Serial.println("followWallBehavior");

  while (steppers.run()) {
    lidar_data = RPC.call("read_lidars").as<struct lidar>();
    if (leftHasWall() && !rightHasWall() && lidar_data.left > DEADBAND_INNER_CM) {
      // only left has wall
      followLeft();
      break;
    } else if (!leftHasWall() && rightHasWall() && lidar_data.left > DEADBAND_INNER_CM) {
      // only right has wall
      followRight();
      break;
    } else if (leftHasWall() && rightHasWall()) {
      // both left and right have walls
      followCenter();
      break;
    } else if (frontHasWall()) {
      spin(TO_LEFT, 90, defaultStepSpeed);
      break;
    } else if (lidar_data.right < DEADBAND_INNER_CM || lidar_data.left < DEADBAND_INNER_CM) {
      avoidObstacle();
    }
  }
}

/**
 * @brief Function to follow the left wall using lidar data.
 *
 * This function sets the speed of the left and right stepper motors to a base speed,
 * updates the current state to FOLLOWING_LEFT, and then enters an infinite loop to
 * continuously adjust the robot's path based on lidar readings.
 *
 * The function performs the following steps:
 * 1. Sets the speed of both stepper motors to FOLLOW_WALL_BASE_SPEED.
 * 2. Updates the current state to FOLLOWING_LEFT and updates the LEDs.
 * 3. Enters an infinite loop to continuously read lidar data and adjust the robot's path.
 * 4. Checks if there is a wall on the right side and calls followCenter() if both sides have walls.
 * 5. Detects inner and outer corners on the left side.
 * 6. Calculates the error between the current distance to the left wall and the target distance.
 * 7. Adjusts the speed of the motors based on the error to maintain the desired distance from the wall.
 * 8. Updates the current state and LEDs based on the error.
 * 9. Runs the motors at the adjusted speeds.
 *
 * The function uses a proportional-derivative (PD) controller to adjust the motor speeds
 * based on the error and its derivative.
```

```
 *
 * @note This function contains an infinite loop and will not return.
 */
void followLeft() {
  stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
  stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);

  currentState = FOLLOWING_LEFT;
  updateLEDs();
  delay(1000);

  while (true) {
    lidar_data = RPC.call("read_lidars").as<struct lidar>();

    if (rightHasWall()) {
      // both left and right have walls
      followCenter();
    }

    // check if there is a corner on the left (inner corner)
    detectLeftInnerCorner();

    // check if there is a corner on the right (outer corner)
    if (!leftHasWall() && !detectLeftOuterCorner())
      return;

    // Calculate error (how far we are from desired distance)
    double error = lidar_data.left - TARGET_DISTANCE_CM;
    double error_diff = error - lastError;

    // Store current values for next iteration
    lastError = error;

    // If within deadband, drive straight
    if (lidar_data.left >= DEADBAND_INNER_CM && lidar_data.left <= DEADBAND_OUTER_CM) {
      currentState = INSIDE_DEADBAND;
      updateLEDs();
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
    } else if (lidar_data.left < DEADBAND_INNER_CM) {
      avoidObstacle();
    } else {
      // Calculate speed adjustment based on error
      int speedAdjustment = (int)(WallFollowKp * error + WallFollowKd * error_diff);

      // If too far from wall (positive error)
      if (error > 0) {
```

```
      currentState = TOO_FAR;
      updateLEDs();

      // Turn towards wall - slow down let motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
    }
    // If too close to wall (negative error)
    else {
      currentState = TOO_CLOSE;
      updateLEDs();

      // Turn away from wall - slow down rigt motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
    }
   }

   stepperLeft.runSpeed();
   stepperRight.runSpeed();
  }
}

/**
 * @brief Detects if the robot is at the left inner corner and updates the state accordingly.
 *
 * This function checks if the robot is at the left inner corner using the isLeftCorner() function.
 * If the robot is at the left inner corner, it updates the current state to INSIDE_CORNER,
 * updates the LEDs to reflect the new state, and then spins the robot 90 degrees to the right
 * at the default step speed.
 */
void detectLeftInnerCorner() {
  if (isLeftCorner()) {
    currentState = INSIDE_CORNER;
    updateLEDs();

    spin(TO_RIGHT, 90, defaultStepSpeed);
  }
}

/**
 * @brief Detects the left outer corner of the robot's path.
 *
 * This function performs a series of movements to detect the left outer corner.
 * It updates the current state to OUTSIDE_CORNER and updates the LEDs accordingly.
 * The robot moves forward, spins to the left, and moves forward again.
 * It then reads the LIDAR data to determine if there is a wall on the left side.
```

```
 *
 * @return true if there is a wall on the left side, false otherwise.
 */
bool detectLeftOuterCorner() {
  currentState = OUTSIDE_CORNER;
  updateLEDs();

  forward(2, FOLLOW_WALL_BASE_SPEED);
  spin(TO_LEFT, 90, FOLLOW_WALL_BASE_SPEED);
  forward(12, FOLLOW_WALL_BASE_SPEED);

  lidar_data = RPC.call("read_lidars").as<struct lidar>();

  return leftHasWall();
}

/**
 * @brief Function to follow the right wall using LIDAR data and stepper motors.
 *
 * This function sets the speed of the left and right stepper motors to a base speed
 * and enters a loop where it continuously reads LIDAR data to adjust the robot's
 * movement to follow the right wall. The function uses proportional control to
 * maintain a target distance from the wall and handles different scenarios such as
 * detecting corners and avoiding obstacles.
 *
 * The function performs the following steps:
 * 1. Sets the initial speed of the stepper motors.
 * 2. Sets the current state to FOLLOWING_RIGHT and updates the LEDs.
 * 3. Enters an infinite loop to continuously read LIDAR data and adjust motor speeds.
 * 4. Checks if there is a wall on the left and calls followCenter() if both left and right have walls.
 * 5. Detects inner and outer corners on the right.
 * 6. Calculates the error between the current distance and the target distance.
 * 7. Applies proportional control to adjust motor speeds based on the error.
 * 8. Updates the current state and LEDs based on the error.
 * 9. Runs the stepper motors at the adjusted speeds.
 *
 * @note This function assumes the existence of several global variables and functions:
 * - stepperLeft, stepperRight: Instances of stepper motor control.
 * - FOLLOW_WALL_BASE_SPEED: Base speed for the motors.
 * - currentState: Current state of the robot.
 * - updateLEDs(): Function to update the state of LEDs.
 * - lidar_data: Struct containing LIDAR data.
 * - TARGET_DISTANCE_CM: Desired distance from the wall.
 * - DEADBAND_INNER_CM, DEADBAND_OUTER_CM: Deadband range for distance control.
 * - WallFollowKp, WallFollowKd: Proportional and derivative gains for control.
 * - lastError: Previous error value for derivative control.
 * - leftHasWall(), rightHasWall(): Functions to check for walls on the left and right.
```

```
 * - detectRightInnerCorner(), detectRightOuterCorner(): Functions to detect corners.
 * - avoidObstacle(): Function to avoid obstacles.
 */
void followRight() {
  stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
  stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);

  currentState = FOLLOWING_RIGHT;
  updateLEDs();
  delay(1000);

  while (true) {

    lidar_data = RPC.call("read_lidars").as<struct lidar>();

    if (leftHasWall()) {
      // both left and right have walls
      followCenter();
    }

    // check if there is a corner on the right (inner corner)
    detectRightInnerCorner();

    // check if there is a corner on the right (outer corner)
    if (!rightHasWall() && !detectRightOuterCorner())
      return;

    // Calculate error (how far we are from desired distance)
    double error = lidar_data.right - TARGET_DISTANCE_CM;
    double error_diff = error - lastError;

    // Store current values for next iteration
    lastError = error;

    // If within deadband, drive straight
    if (lidar_data.right >= DEADBAND_INNER_CM && lidar_data.right <= DEADBAND_OUTER_CM) {
      currentState = INSIDE_DEADBAND;
      updateLEDs();
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
    }  // Outside deadband - apply proportional control
    else if (lidar_data.right < DEADBAND_INNER_CM) {
      avoidObstacle();
    } else {
      // Calculate speed adjustment based on error
      int speedAdjustment = (int)(WallFollowKp * error + WallFollowKd * error_diff);
```

```
    // If too far from wall (positive error)
    if (error > 0) {
      currentState = TOO_FAR;
      updateLEDs();
      // Turn towards wall - slow down right motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
    }

    // If too close to wall (negative error)
    else {
      currentState = TOO_CLOSE;
      updateLEDs();
      // Turn away from wall - slow down left motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
    }
   }

   stepperLeft.runSpeed();
   stepperRight.runSpeed();
  }
}

/**
 * @brief Detects if the robot is at a right inner corner and updates its state accordingly.
 *
 * This function checks if the robot is at a right inner corner using the isRightCorner() function.
 * If a right inner corner is detected, it updates the current state to INSIDE_CORNER, updates the LEDs,
 * and makes the robot spin to the left by 90 degrees at the default step speed.
 */
void detectRightInnerCorner() {
  if (isRightCorner()) {
    currentState = INSIDE_CORNER;
    updateLEDs();

    spin(TO_LEFT, 90, defaultStepSpeed);
  }
}

/**
 * @brief Detects the right outer corner of the environment.
 *
 * This function performs a series of movements to detect the right outer corner.
 * It updates the current state to OUTSIDE_CORNER and updates the LEDs accordingly.
 * The robot moves forward a short distance, spins to the right by 90 degrees,
 * and then moves forward a longer distance. It then reads the LIDAR data to
```

```
 * determine if there is a wall on the right side.
 *
 * @return true if there is a wall on the right side, false otherwise.
 */
bool detectRightOuterCorner() {
  currentState = OUTSIDE_CORNER;
  updateLEDs();

  forward(2, FOLLOW_WALL_BASE_SPEED);
  spin(TO_RIGHT, 90, FOLLOW_WALL_BASE_SPEED);
  forward(12, FOLLOW_WALL_BASE_SPEED);

  lidar_data = RPC.call("read_lidars").as<struct lidar>();

  return rightHasWall();
}

/**
 * @brief Function to follow the center path between two walls using LIDAR data.
 *
 * This function sets the speed of the left and right stepper motors to a base speed
 * and continuously adjusts the speed based on the LIDAR readings to maintain a central
 * position between two walls. It updates the current state to FOLLOWING_CENTER and
 * updates the LEDs accordingly.
 *
 * The function reads LIDAR data to determine the presence of walls on the left, right,
 * and front. Depending on the presence of walls, it calls appropriate functions to
 * follow the left or right wall, or to turn around if a wall is detected in front.
 *
 * The center error is calculated as the difference between the left and right LIDAR
 * readings. A speed adjustment is computed based on the error and its difference from
 * the last error, and the speeds of the stepper motors are adjusted accordingly.
 *
 * The function runs indefinitely in a loop, continuously adjusting the motor speeds
 * to follow the center path.
 *
 * @note The function assumes the existence of certain global variables and functions:
 * - FOLLOW_WALL_BASE_SPEED: Base speed for the motors.
 * - currentState: Variable to store the current state of the robot.
 * - updateLEDs(): Function to update the LEDs based on the current state.
 * - RPC.call("read_lidars"): Function to read LIDAR data.
 * - leftHasWall(), rightHasWall(), frontHasWall(): Functions to check for walls.
 * - followLeft(), followRight(): Functions to follow the left or right wall.
 * - goToAngle(int angle): Function to turn the robot to a specified angle.
 * - WallFollowKd: Constant for the derivative term in the speed adjustment calculation.
 * - lastError: Variable to store the last error value.
 * - lidar_data: Struct to store LIDAR readings.
```

```
 */
void followCenter() {
  stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
  stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
  currentState = FOLLOWING_CENTER;
  updateLEDs();

  while (true) {

    lidar_data = RPC.call("read_lidars").as<struct lidar>();

    if (leftHasWall() && !rightHasWall()) {
      followLeft();
    } else if (!leftHasWall() && rightHasWall()) {
      followRight();
    } else if (!leftHasWall() && !rightHasWall()) {
      return;
    } else if (leftHasWall() && rightHasWall() && frontHasWall()) {
      goToAngle(180);
    }

    // Calculate center error (positive means closer to left wall)
    float error = lidar_data.left - lidar_data.right;
    float error_diff = error - lastError;

    double speedAdjustment = 0.9 * error + WallFollowKd * error_diff;

    if (abs(error) <= 3) {
      speedAdjustment = 0;
    }

    stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED - speedAdjustment);
    stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED + speedAdjustment);

    stepperLeft.runSpeed();
    stepperRight.runSpeed();

    lastError = error;
  }
}

enum ToGoalState {
  MOVING,
  AVOID_OBSTACLE_1,
  AVOID_OBSTACLE_2,
  AVOID_OBSTACLE_3,
};
```

```
/**
 * @brief Sets the position of both the left and right stepper motor encoders.
 *
 * This function updates the encoder positions for both the left and right stepper motors
 * to the specified position.
 *
 * @param position The new position to set for both the left and right encoders.
 */
void setStepperCounter(long position) {
  encoder[LEFT] = position;
  encoder[RIGHT] = position;
}

/**
 * @brief Moves the robot to a specified goal while avoiding obstacles.
 *
 * This function calculates the angle and distance to the goal, then moves the robot
 * towards the goal while checking for obstacles using LIDAR data. If an obstacle is detected,
 * the robot will perform a series of maneuvers to avoid the obstacle and then continue towards
 * the goal.
 *
 * @param x The x-coordinate of the goal.
 * @param y The y-coordinate of the goal.
 *
 * The function uses the following states to manage the robot's movement:
 * - MOVING: The robot is moving towards the goal.
 * - AVOID_OBSTACLE_1: The robot detected an obstacle and is performing the first avoidance
maneuver.
 * - AVOID_OBSTACLE_2: The robot is performing the second avoidance maneuver.
 * - AVOID_OBSTACLE_3: The robot is performing the third avoidance maneuver and will resume moving
towards the goal.
 *
 * The function uses LIDAR data to detect obstacles and encoder data to track the robot's movement.
 * It adjusts the robot's speed and direction based on the current state and obstacle detection.
 *
 * The function also includes debug prints to the Serial monitor to indicate the current state and actions.
 */
void goToGoalAvoidbs(double x, double y) {

  int obsCount = 0;

  double angle = getTurnAngle(x, y);

  goToAngle(angle);

  double distanceSteps = length2Steps(sqrt(pow(x, 2) + pow(y, 2)));
```

```
 Serial.println(distanceSteps);

 int eCounts = 0;
 int avoidObsCount = 0;
 setStepperCounter(0);

 ToGoalState state = MOVING;

 while (eCounts * encoderRatio < distanceSteps) {
  lidar_data = RPC.call("read_lidars").as<struct lidar>();

  switch (state) {
    case MOVING:
      if (lidar_data.front < 10) {
        spin(TO_LEFT, 90, defaultStepSpeed);
        state = AVOID_OBSTACLE_1;
        setStepperCounter(0);
      } else {
        eCounts = encoder[LEFT];
        Serial.println("MOVING");
      }
      break;
    case AVOID_OBSTACLE_1:
      if (lidar_data.right > 30) {
        forward(2, defaultStepSpeed);
        spin(TO_RIGHT, 90, defaultStepSpeed);
        forward(14, defaultStepSpeed);
        state = AVOID_OBSTACLE_2;
        setStepperCounter(eCounts);
      } else {
        avoidObsCount = encoder[LEFT];
        Serial.println("AVOID 1");
      }
      break;
    case AVOID_OBSTACLE_2:
      if (lidar_data.right > 30) {
        forward(4, defaultStepSpeed);
        spin(TO_RIGHT, 90, defaultStepSpeed);
        forward(5, defaultStepSpeed);
        state = AVOID_OBSTACLE_3;
        setStepperCounter(0);
      } else {
        eCounts = encoder[LEFT];
        Serial.println("AVOID 2");
      }
      break;
    case AVOID_OBSTACLE_3:
```

```
      if (encoder[LEFT] >= avoidObsCount) {
        spin(TO_LEFT, 85, defaultStepSpeed);
        state = MOVING;
        eCounts += length2Steps(14) / encoderRatio;
        setStepperCounter(eCounts);
      } else {
        Serial.println("AVOID 3");
      }
      break;
    default:
      break;
  }

  stepperLeft.setSpeed(defaultStepSpeed);
  stepperRight.setSpeed(defaultStepSpeed);
  stepperLeft.runSpeed();
  stepperRight.runSpeed();
 }

 delay(10000);
}

/**
 * @brief Initializes the stepper motor and sets up interrupts for the left and right wheel encoders.
 *
 * This function performs the following tasks:
 * - Initializes the stepper motor by calling init_stepper().
 * - Attaches an interrupt to the left wheel encoder pin, triggering the LwheelSpeed function on any
change.
 * - Attaches an interrupt to the right wheel encoder pin, triggering the RwheelSpeed function on any
change.
 * - Delays execution for 1000 milliseconds to allow for setup stabilization.
 */
void setupM7() {
 init_stepper();                                    // set up stepper motor
 attachInterrupt(digitalPinToInterrupt(ltEncoder), LwheelSpeed, CHANGE);  // init the interrupt mode
for the left encoder
 attachInterrupt(digitalPinToInterrupt(rtEncoder), RwheelSpeed, CHANGE);  // init the interrupt mode
for the right encoder
 delay(1000);
}

/**
 * @brief Main loop function for behavior control.
 *
 * This function initializes the state of three LEDs (red, yellow, green) to off.
 * It then enters an infinite loop where it continuously executes the
```

```
 * smartFollowBehavior function. Other behaviors such as smartWanderBehavior
 * and runawayBehavior are commented out and can be enabled if needed.
 */
void loopM7() {
  digitalWrite(redLED, LOW);  // Initially LED is off
  digitalWrite(ylwLED, LOW);
  digitalWrite(grnLED, LOW);
  while (true) {
    // smartWanderBehavior();
    // smartFollowBehavior();
    // runawayBehavior();
    // followWallBehavior();
    goToGoalAvoidbs(72.0, 0.0);
  }
}

// set up the M4 to be the server for the sensors data
void setupM4() {
  for (int i = 0; i < numOfSens; i++) {
    pinMode(lidar_pins[i], OUTPUT);
  }
  RPC.bind("read_lidars", read_lidars);  // bind a method to return the lidar data all at once
  RPC.bind("read_sonars", read_sonars);  // bind a method to return the lidar data all at once
}

// loop for the M4 to read the sensors
void loopM4() {
  // Add delays between readings to allow sensor to stabilize
  lidarData.front = read_lidar(ft_lidar);
  // delay(10);
  lidarData.back = read_lidar(bk_lidar);
  // delay(10);
  lidarData.left = read_lidar(lt_lidar);
  // delay(10);
  lidarData.right = read_lidar(rt_lidar);
  // delay(10);

  // sonarData.right = read_sonar(leftSnr);
  // delay(20);
  // sonarData.left = read_sonar(rightSnr);
  // delay(20);
}

// MAIN
void setup() {
  int baudrate = 9600;       // serial monitor baud rate'
  randomSeed(analogRead(0));  // generate a new random number each time called
```

```
  Serial.begin(baudrate);    // start serial monitor communication
  Serial.println("Robot starting...Put ON TEST STAND");

  RPC.begin();
  if (HAL_GetCurrentCPUID() == CM7_CPUID) {
    // if on M7 CPU, run M7 setup & loop
    setupM7();
    while (1)
      loopM7();
  } else {
    // if on M4 CPU, run M7 setup & loop
    setupM4();
    while (1)
      loopM4();
  }
}

void loop() {}
```